**Exercise 1: Employee Management System - Overview and Setup**

**1. Creating a Spring Boot Project**

To set up a Spring Boot project, follow these instructions:

1. **Using Spring Initializr:**
   * Open [Spring Initializr](https://start.spring.io/) in your web browser.
   * Complete the project details:
     + **Project**: Maven Project
     + **Language**: Java
     + **Spring Boot**: Select the latest stable release
     + **Project Metadata**:
       - Group: ‘com.example’
       - Artifact: ‘EmployeeManagementSystem’
       - Name: ‘EmployeeManagementSystem’
       - Package Name: ‘com.example.employeemanagementsystem’
       - Packaging: Jar
       - Java: 17 or higher
   * **Add Dependencies**:
     + Spring Web
     + Spring Data JPA
     + H2 Database
     + Lombok
   * Click on **Generate** to download the zip file containing the project.
   * Unzip the file and import the project into your IDE (such as IntelliJ IDEA or Eclipse).

**2. Configuring Application Properties**

Modify the ‘src/main/resources/application.properties’ file to set up the H2 database connection:

# H2 Database settings

spring.datasource.url=jdbc:h2:mem:testdb

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=password

spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

# Enable H2 Console for easier access (optional for debugging purposes)

spring.h2.console.enabled=true

spring.h2.console.path=/h2-console

# Hibernate JPA configuration

spring.jpa.hibernate.ddl-auto=update

**Exercise 2: Employee Management System - Creating Entities**

**1. Defining JPA Entities**

Create the ‘Employee’ and ‘Department’ entities in the ‘com.example.employeemanagementsystem.model’ package.

1. **Employee Entity:**

package com.example.employeemanagementsystem.model;

import jakarta.persistence.\*;

import lombok.Data;

@Data

@Entity

@Table(name = "employees")

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String email;

@ManyToOne

@JoinColumn(name = "department\_id")

private Department department;

}

1. Department Entity:

package com.example.employeemanagementsystem.model;

import jakarta.persistence.\*;

import lombok.Data;

import java.util.List;

@Data

@Entity

@Table(name = "departments")

public class Department {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

@OneToMany(mappedBy = "department", cascade = CascadeType.ALL, fetch = FetchType.LAZY)

private List<Employee> employees;

}

**Defining Repositories**

Create JPA repositories for performing CRUD operations on the entities.

1. **Employee Repository:**

package com.example.employeemanagementsystem.repository;

import com.example.employeemanagementsystem.model.Employee;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;

@Repository

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

}

1. Department Repository:

package com.example.employeemanagementsystem.repository;

import com.example.employeemanagementsystem.model.Department;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;

@Repository

public interface DepartmentRepository extends JpaRepository<Department, Long> {

}

**Implementing Services**

Create services to encapsulate business logic for the entities.

1. **Employee Service:**

package com.example.employeemanagementsystem.service;

import com.example.employeemanagementsystem.model.Employee;

import com.example.employeemanagementsystem.repository.EmployeeRepository;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import java.util.List;

import java.util.Optional;

@Service

public class EmployeeService {

@Autowired

private EmployeeRepository employeeRepository;

public List<Employee> getAllEmployees() {

return employeeRepository.findAll();

}

public Optional<Employee> getEmployeeById(Long id) {

return employeeRepository.findById(id);

}

public Employee saveEmployee(Employee employee) {

return employeeRepository.save(employee);

}

public void deleteEmployee(Long id) {

employeeRepository.deleteById(id);

}

}

1. Department Service:

package com.example.employeemanagementsystem.service;

import com.example.employeemanagementsystem.model.Department;

import com.example.employeemanagementsystem.repository.DepartmentRepository;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import java.util.List;

import java.util.Optional;

@Service

public class DepartmentService {

@Autowired

private DepartmentRepository departmentRepository;

public List<Department> getAllDepartments() {

return departmentRepository.findAll();

}

public Optional<Department> getDepartmentById(Long id) {

return departmentRepository.findById(id);

}

public Department saveDepartment(Department department) {

return departmentRepository.save(department);

}

public void deleteDepartment(Long id) {

departmentRepository.deleteById(id);

}

}

**Creating Controllers**

Create REST controllers to define the API endpoints for managing employees and departments.

1. **Employee Controller:**

package com.example.employeemanagementsystem.controller;

import com.example.employeemanagementsystem.model.Employee;

import com.example.employeemanagementsystem.service.EmployeeService;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import java.util.List;

import java.util.Optional;

@RestController

@RequestMapping("/employees")

public class EmployeeController {

@Autowired

private EmployeeService employeeService;

@GetMapping

public List<Employee> getAllEmployees() {

return employeeService.getAllEmployees();

}

@GetMapping("/{id}")

public ResponseEntity<Employee> getEmployeeById(@PathVariable Long id) {

Optional<Employee> employee = employeeService.getEmployeeById(id);

return employee.map(ResponseEntity::ok).orElseGet(() -> ResponseEntity.notFound().build());

}

@PostMapping

public Employee createEmployee(@RequestBody Employee employee) {

return employeeService.saveEmployee(employee);

}

@PutMapping("/{id}")

public ResponseEntity<Employee> updateEmployee(@PathVariable Long id, @RequestBody Employee employeeDetails) {

Optional<Employee> employee = employeeService.getEmployeeById(id);

if (employee.isPresent()) {

Employee updatedEmployee = employee.get();

updatedEmployee.setName(employeeDetails.getName());

updatedEmployee.setEmail(employeeDetails.getEmail());

updatedEmployee.setDepartment(employeeDetails.getDepartment());

return ResponseEntity.ok(employeeService.saveEmployee(updatedEmployee));

} else {

return ResponseEntity.notFound().build();

}

}

@DeleteMapping("/{id}")

public ResponseEntity<Void> deleteEmployee(@PathVariable Long id) {

employeeService.deleteEmployee(id);

return ResponseEntity.noContent().build();

}

}

1. Department Controller:

package com.example.employeemanagementsystem.controller;

import com.example.employeemanagementsystem.model.Department;

import com.example.employeemanagementsystem.service.DepartmentService;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import java.util.List;

import java.util.Optional;

@RestController

@RequestMapping("/departments")

public class DepartmentController {

@Autowired

private DepartmentService departmentService;

@GetMapping

public List<Department> getAllDepartments() {

return departmentService.getAllDepartments();

}

@GetMapping("/{id}")

public ResponseEntity<Department> getDepartmentById(@PathVariable Long id) {

Optional<Department> department = departmentService.getDepartmentById(id);

return department.map(ResponseEntity::ok).orElseGet(() -> ResponseEntity.notFound().build());

}

@PostMapping

public Department createDepartment(@RequestBody Department department) {

return departmentService.saveDepartment(department);

}

@PutMapping("/{id}")

public ResponseEntity<Department> updateDepartment(@PathVariable Long id, @RequestBody Department departmentDetails) {

Optional<Department> department = departmentService.getDepartmentById(id);

if (department.isPresent()) {

Department updatedDepartment = department.get();

updatedDepartment.setName(departmentDetails.getName());

return ResponseEntity.ok(departmentService.saveDepartment(updatedDepartment));

} else {

return ResponseEntity.notFound().build();

}

}

@DeleteMapping("/{id}")

public ResponseEntity<Void> deleteDepartment(@PathVariable Long id) {

departmentService.deleteDepartment(id);

return ResponseEntity.noContent().build();

}

}

**Running and Testing the Application**

1. **Starting the Application:**
   * Start the application by running the main method in the EmployeeManagementSystemApplication class.
2. **Accessing the H2 Console:**
   * Visit http://localhost:8080/h2-console to use the H2 database console.
   * Use the following login details:
     + **JDBC URL**: jdbc:h2:mem:testdb
     + **Username**: sa
     + **Password**: password
3. **Testing the API Endpoints:**
   * You can test the endpoints using tools like Postman or cURL:
     + GET /employees: Fetch all employees.
     + GET /employees/{id}: Get employee details by ID.
     + POST /employees: Add a new employee.
     + PUT /employees/{id}: Modify an existing employee.
     + DELETE /employees/{id}: Remove an employee.
     + GET /departments: Fetch all departments.
     + GET /departments/{id}: Get department details by ID.
     + POST /departments: Add a new department.
     + PUT /departments/{id}: Modify an existing department.
     + DELETE /departments/{id}: Remove a department.

This setup provides you with an Employee Management System with essential CRUD operations for both employees and departments.

**Exercise 3: Employee Management System - Creating Repositories**

**1. Introduction to Spring Data Repositories**

**Advantages of using Spring Data Repositories:**

* **Ease of Use**: Spring Data repositories minimize boilerplate code by providing built-in methods for basic CRUD operations.
* **Uniformity**: Implementing repository interfaces ensures consistent data access patterns across the application.
* **Automatic Query Generation**: Spring Data allows you to create custom queries just by defining method signatures in the repository interfaces.
* **Built-in Pagination and Sorting**: The repository interfaces include support for paginating and sorting query results.

**2. Defining Repositories**

Create interfaces for ‘EmployeeRepository’ and ‘DepartmentRepository’ by extending JpaRepository.

1. **Employee Repository:**

package com.example.employeemanagementsystem.repository;

import com.example.employeemanagementsystem.model.Employee;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;

import java.util.List;

@Repository

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

// Automatically generated query to fetch employees by department name

List<Employee> findByDepartmentName(String departmentName);

// Automatically generated query to search employees by name (case insensitive)

List<Employee> findByNameContainingIgnoreCase(String name);

}

1. Department Repository:

package com.example.employeemanagementsystem.repository;

import com.example.employeemanagementsystem.model.Department;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;

@Repository

public interface DepartmentRepository extends JpaRepository<Department, Long> {

// Automatically generated query to find a department by its name

Department findByName(String name);

}

**Exercise 4: Employee Management System - Implementing CRUD Operations**

**1. Implementing Basic CRUD Operations**

Make use of JpaRepository methods to handle the creation, retrieval, updating, and deletion of employees and departments. We will also expose REST endpoints for these operations via EmployeeController and DepartmentController.

1. **Employee Service**

**package com.example.employeemanagementsystem.service;**

**import com.example.employeemanagementsystem.model.Employee;**

**import com.example.employeemanagementsystem.repository.EmployeeRepository;**

**import org.springframework.beans.factory.annotation.Autowired;**

**import org.springframework.stereotype.Service;**

**import java.util.List;**

**import java.util.Optional;**

**@Service**

**public class EmployeeService {**

**@Autowired**

**private EmployeeRepository employeeRepository;**

**// Fetch all employees from the repository**

**public List<Employee> getAllEmployees() {**

**return employeeRepository.findAll();**

**}**

**// Find an employee by their ID**

**public Optional<Employee> getEmployeeById(Long id) {**

**return employeeRepository.findById(id);**

**}**

**// Save a new or updated employee to the repository**

**public Employee saveEmployee(Employee employee) {**

**return employeeRepository.save(employee);**

**}**

**// Update an existing employee using their ID and new details**

**public Employee updateEmployee(Long id, Employee employeeDetails) {**

**return employeeRepository.findById(id).map(employee -> {**

**employee.setName(employeeDetails.getName());**

**employee.setEmail(employeeDetails.getEmail());**

**employee.setDepartment(employeeDetails.getDepartment());**

**return employeeRepository.save(employee);**

**}).orElseThrow(() -> new RuntimeException("Employee not found with id " + id));**

**}**

**// Delete an employee by their ID**

**public void deleteEmployee(Long id) {**

**employeeRepository.deleteById(id);**

**}**

**// Retrieve employees by department name**

**public List<Employee> getEmployeesByDepartmentName(String departmentName) {**

**return employeeRepository.findByDepartmentName(departmentName);**

**}**

**// Search for employees by name (case insensitive)**

**public List<Employee> searchEmployeesByName(String name) {**

**return employeeRepository.findByNameContainingIgnoreCase(name);**

**}**

**}**

1. **Department Service**

**package com.example.employeemanagementsystem.service;**

**import com.example.employeemanagementsystem.model.Department;**

**import com.example.employeemanagementsystem.repository.DepartmentRepository;**

**import org.springframework.beans.factory.annotation.Autowired;**

**import org.springframework.stereotype.Service;**

**import java.util.List;**

**import java.util.Optional;**

**@Service**

**public class DepartmentService {**

**@Autowired**

**private DepartmentRepository departmentRepository;**

**// Fetch all departments from the repository**

**public List<Department> getAllDepartments() {**

**return departmentRepository.findAll();**

**}**

**// Find a department by its ID**

**public Optional<Department> getDepartmentById(Long id) {**

**return departmentRepository.findById(id);**

**}**

**// Save a new or updated department to the repository**

**public Department saveDepartment(Department department) {**

**return departmentRepository.save(department);**

**}**

**// Update an existing department using its ID and new details**

**public Department updateDepartment(Long id, Department departmentDetails) {**

**return departmentRepository.findById(id).map(department -> {**

**department.setName(departmentDetails.getName());**

**return departmentRepository.save(department);**

**}).orElseThrow(() -> new RuntimeException("Department not found with id " + id));**

**}**

**// Delete a department by its ID**

**public void deleteDepartment(Long id) {**

**departmentRepository.deleteById(id);**

**}**

**// Retrieve a department by its name**

**public Department getDepartmentByName(String name) {**

**return departmentRepository.findByName(name);**

**}**

**}**

**2. Exposing RESTful Endpoints**

1. **Employee Controller**

**package com.example.employeemanagementsystem.controller;**

**import com.example.employeemanagementsystem.model.Employee;**

**import com.example.employeemanagementsystem.service.EmployeeService;**

**import org.springframework.beans.factory.annotation.Autowired;**

**import org.springframework.http.ResponseEntity;**

**import org.springframework.web.bind.annotation.\*;**

**import java.util.List;**

**import java.util.Optional;**

**@RestController**

**@RequestMapping("/employees")**

**public class EmployeeController {**

**@Autowired**

**private EmployeeService employeeService;**

**// Get a list of all employees**

**@GetMapping**

**public List<Employee> getAllEmployees() {**

**return employeeService.getAllEmployees();**

**}**

**// Get details of an employee by ID**

**@GetMapping("/{id}")**

**public ResponseEntity<Employee> getEmployeeById(@PathVariable Long id) {**

**Optional<Employee> employee = employeeService.getEmployeeById(id);**

**return employee.map(ResponseEntity::ok).orElseGet(() -> ResponseEntity.notFound().build());**

**}**

**// Create a new employee**

**@PostMapping**

**public Employee createEmployee(@RequestBody Employee employee) {**

**return employeeService.saveEmployee(employee);**

**}**

**// Update an existing employee by ID**

**@PutMapping("/{id}")**

**public ResponseEntity<Employee> updateEmployee(@PathVariable Long id, @RequestBody Employee employeeDetails) {**

**try {**

**Employee updatedEmployee = employeeService.updateEmployee(id, employeeDetails);**

**return ResponseEntity.ok(updatedEmployee);**

**} catch (RuntimeException e) {**

**return ResponseEntity.notFound().build();**

**}**

**}**

**// Delete an employee by ID**

**@DeleteMapping("/{id}")**

**public ResponseEntity<Void> deleteEmployee(@PathVariable Long id) {**

**employeeService.deleteEmployee(id);**

**return ResponseEntity.noContent().build();**

**}**

**// Search for employees by name**

**@GetMapping("/search")**

**public List<Employee> searchEmployeesByName(@RequestParam String name) {**

**return employeeService.searchEmployeesByName(name);**

**}**

**// Get a list of employees by department name**

**@GetMapping("/department/{departmentName}")**

**public List<Employee> getEmployeesByDepartment(@PathVariable String departmentName) {**

**return employeeService.getEmployeesByDepartmentName(departmentName);**

**}**

**}**

1. **Department Controller**

**package com.example.employeemanagementsystem.controller;**

**import com.example.employeemanagementsystem.model.Department;**

**import com.example.employeemanagementsystem.service.DepartmentService;**

**import org.springframework.beans.factory.annotation.Autowired;**

**import org.springframework.http.ResponseEntity;**

**import org.springframework.web.bind.annotation.\*;**

**import java.util.List;**

**import java.util.Optional;**

**@RestController**

**@RequestMapping("/departments")**

**public class DepartmentController {**

**@Autowired**

**private DepartmentService departmentService;**

**// Get a list of all departments**

**@GetMapping**

**public List<Department> getAllDepartments() {**

**return departmentService.getAllDepartments();**

**}**

**// Get details of a department by ID**

**@GetMapping("/{id}")**

**public ResponseEntity<Department> getDepartmentById(@PathVariable Long id) {**

**Optional<Department> department = departmentService.getDepartmentById(id);**

**return department.map(ResponseEntity::ok).orElseGet(() -> ResponseEntity.notFound().build());**

**}**

**// Create a new department**

**@PostMapping**

**public Department createDepartment(@RequestBody Department department) {**

**return departmentService.saveDepartment(department);**

**}**

**// Update an existing department by ID**

**@PutMapping("/{id}")**

**public ResponseEntity<Department> updateDepartment(@PathVariable Long id, @RequestBody Department departmentDetails) {**

**try {**

**Department updatedDepartment = departmentService.updateDepartment(id, departmentDetails);**

**return ResponseEntity.ok(updatedDepartment);**

**} catch (RuntimeException e) {**

**return ResponseEntity.notFound().build();**

**}**

**}**

**// Delete a department by ID**

**@DeleteMapping("/{id}")**

**public ResponseEntity<Void> deleteDepartment(@PathVariable Long id) {**

**departmentService.deleteDepartment(id);**

**return ResponseEntity.noContent().build();**

**}**

**// Get details of a department by its name**

**@GetMapping("/name/{name}")**

**public ResponseEntity<Department> getDepartmentByName(@PathVariable String name) {**

**Department department = departmentService.getDepartmentByName(name);**

**if (department != null) {**

**return ResponseEntity.ok(department);**

**} else {**

**return ResponseEntity.notFound().build();**

**}**

**}**

**}**

**Testing the Application**

1. **Run the Application:**
   * **Start the Spring Boot application by running the EmployeeManagementSystemApplication class.**
2. **Testing via Postman or cURL:**
   * **Validate the RESTful endpoints for managing employees and departments:**
     + **GET /employees: Fetch all employees.**
     + **GET /employees/{id}: Fetch a specific employee by ID.**
     + **POST /employees: Create a new employee.**
     + **PUT /employees/{id}: Update an employee by ID.**
     + **DELETE /employees/{id}: Remove an employee by ID.**
     + **GET /employees/search?name={name}: Search for employees by name.**
     + **GET /employees/department/{departmentName}: Fetch employees by department name.**
     + **GET /departments: Fetch all departments.**
     + **GET /departments/{id}: Fetch a specific department by ID.**
     + **POST /departments: Create a new department.**
     + **PUT /departments/{id}: Update a department by ID.**
     + **DELETE /departments/{id}: Remove a department by ID.**
     + **GET /departments/name/{name}: Fetch a department by its name.**

**Exercise 5: Employee Management System - Defining Query Methods**

**1. Implementing Query Methods**

**Custom Query Methods Using Keywords:**

Spring Data JPA supports the creation of query methods based on naming conventions. Let's expand the EmployeeRepository with additional query methods

package com.example.employeemanagementsystem.repository;

import com.example.employeemanagementsystem.model.Employee;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.data.jpa.repository.Query;

import org.springframework.data.repository.query.Param;

import org.springframework.stereotype.Repository;

import java.util.List;

@Repository

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

// Query method to retrieve employees based on the department's name

List<Employee> findByDepartmentName(String departmentName);

// Query method to find employees whose names contain a specified string, case-insensitive

List<Employee> findByNameContainingIgnoreCase(String name);

// Custom query defined with @Query annotation to find an employee by email address

@Query("SELECT e FROM Employee e WHERE e.email = :email")

Employee findEmployeeByEmail(@Param("email") String email);

// Custom query method to get employees by the department's ID using JPQL

@Query("SELECT e FROM Employee e WHERE e.department.id = :departmentId")

List<Employee> findByDepartmentId(@Param("departmentId") Long departmentId);

}

**2. Using Named Queries**

Named queries are pre-defined at the entity level, allowing their reuse throughout the application. Here's how to define and use them:

**Define Named Queries:**

package com.example.employeemanagementsystem.model;

import jakarta.persistence.\*;

@Entity

@Table(name = "employees")

@NamedQueries({

@NamedQuery(name = "Employee.findByDepartmentNameNamedQuery",

query = "SELECT e FROM Employee e WHERE e.department.name = :departmentName"),

@NamedQuery(name = "Employee.findByEmailNamedQuery",

query = "SELECT e FROM Employee e WHERE e.email = :email")

})

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String email;

@ManyToOne

@JoinColumn(name = "department\_id")

private Department department;

// Getters and setters...

}

**Execute Named Queries:**

To execute named queries, use EntityManager:

package com.example.employeemanagementsystem.service;

import com.example.employeemanagementsystem.model.Employee;

import com.example.employeemanagementsystem.repository.EmployeeRepository;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import jakarta.persistence.EntityManager;

import jakarta.persistence.PersistenceContext;

import jakarta.persistence.TypedQuery;

import java.util.List;

import java.util.Optional;

@Service

public class EmployeeService {

@Autowired

private EmployeeRepository employeeRepository;

@PersistenceContext

private EntityManager entityManager;

// Retrieve a list of employees using the named query for department name

public List<Employee> getEmployeesByDepartmentNameNamedQuery(String departmentName) {

TypedQuery<Employee> query = entityManager.createNamedQuery("Employee.findByDepartmentNameNamedQuery", Employee.class);

query.setParameter("departmentName", departmentName);

return query.getResultList();

}

// Find a single employee using the named query for email

public Employee findEmployeeByEmailNamedQuery(String email) {

TypedQuery<Employee> query = entityManager.createNamedQuery("Employee.findByEmailNamedQuery", Employee.class);

query.setParameter("email", email);

return query.getSingleResult();

}

// Other service methods...

}

**Exercise 6: Employee Management System - Implementing Pagination and Sorting**

**1. Pagination**

To enable pagination, use the Page and Pageable interfaces provided by Spring Data JPA.

**Repository Update:**

package com.example.employeemanagementsystem.repository;

import com.example.employeemanagementsystem.model.Employee;

import org.springframework.data.domain.Page;

import org.springframework.data.domain.Pageable;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;

@Repository

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

// Other query methods...

// Method to retrieve a paginated list of employees

Page<Employee> findAll(Pageable pageable);

}

Service Method for Pagination:

package com.example.employeemanagementsystem.service;

import com.example.employeemanagementsystem.model.Employee;

import com.example.employeemanagementsystem.repository.EmployeeRepository;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.data.domain.Page;

import org.springframework.data.domain.Pageable;

import org.springframework.stereotype.Service;

@Service

public class EmployeeService {

@Autowired

private EmployeeRepository employeeRepository;

// Retrieve a page of employees based on the pageable configuration

public Page<Employee> getEmployeesWithPagination(Pageable pageable) {

return employeeRepository.findAll(pageable);

}

// Other service methods...

}

Controller Endpoint for Pagination:

package com.example.employeemanagementsystem.controller;

import com.example.employeemanagementsystem.model.Employee;

import com.example.employeemanagementsystem.service.EmployeeService;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.data.domain.Page;

import org.springframework.data.domain.Pageable;

import org.springframework.web.bind.annotation.\*;

@RestController

@RequestMapping("/employees")

public class EmployeeController {

@Autowired

private EmployeeService employeeService;

// Endpoint to get a paginated list of employees

@GetMapping("/page")

public Page<Employee> getAllEmployeesWithPagination(Pageable pageable) {

return employeeService.getEmployeesWithPagination(pageable);

}

// Other endpoints...

}

**2. Sorting**

Implement sorting by utilizing the Sort object.

**Repository Update:**

package com.example.employeemanagementsystem.repository;

import com.example.employeemanagementsystem.model.Employee;

import org.springframework.data.domain.Page;

import org.springframework.data.domain.Pageable;

import org.springframework.data.domain.Sort;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;

@Repository

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

// Other query methods...

// Method to retrieve employees with sorting capabilities

Page<Employee> findAll(Pageable pageable);

List<Employee> findAll(Sort sort);

}

Service Method for Sorting:

package com.example.employeemanagementsystem.service;

import com.example.employeemanagementsystem.model.Employee;

import com.example.employeemanagementsystem.repository.EmployeeRepository;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.data.domain.Page;

import org.springframework.data.domain.Pageable;

import org.springframework.data.domain.Sort;

import org.springframework.stereotype.Service;

import java.util.List;

@Service

public class EmployeeService {

@Autowired

private EmployeeRepository employeeRepository;

// Retrieve a page of employees with sorting applied

public Page<Employee> getEmployeesWithPaginationAndSorting(Pageable pageable) {

return employeeRepository.findAll(pageable);

}

// Retrieve a list of employees with sorting applied

public List<Employee> getEmployeesWithSorting(Sort sort) {

return employeeRepository.findAll(sort);

}

// Other service methods...

}

Controller Endpoint for Pagination and Sorting:

package com.example.employeemanagementsystem.controller;

import com.example.employeemanagementsystem.model.Employee;

import com.example.employeemanagementsystem.service.EmployeeService;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.data.domain.Page;

import org.springframework.data.domain.Pageable;

import org.springframework.data.domain.Sort;

import org.springframework.web.bind.annotation.\*;

import java.util.List;

@RestController

@RequestMapping("/employees")

public class EmployeeController {

@Autowired

private EmployeeService employeeService;

// Endpoint to get a paginated and sorted list of employees

@GetMapping("/page")

public Page<Employee> getAllEmployeesWithPaginationAndSorting(Pageable pageable) {

return employeeService.getEmployeesWithPaginationAndSorting(pageable);

}

// Endpoint to get a sorted list of employees

@GetMapping("/sorted")

public List<Employee> getAllEmployeesWithSorting(Sort sort) {

return employeeService.getEmployeesWithSorting(sort);

}

// Other endpoints...

}

**Testing Pagination and Sorting**

1. **Pagination:**
   * Use the GET /employees/page endpoint with query parameters like ?page=0&size=5 to test pagination.
2. **Sorting:**
   * Use the GET /employees/sorted endpoint with Sort parameters like ?sort=name,asc or ?sort=name,desc to test sorting.
3. **Combined Pagination and Sorting:**
   * Use the GET /employees/page endpoint with parameters like ?page=0&size=5&sort=name,asc to test both pagination and sorting together.

**Exercise 7: Employee Management System - Enabling Entity Auditing**

Entity auditing allows you to monitor who created or updated an entity and when these changes occurred. To set this up, we’ll use Spring Data JPA’s auditing features.

**1. Enable Auditing**

To activate auditing in a Spring Boot application, configure it in your setup and annotate your entity classes accordingly.

**Step 1: Enable Auditing in Configuration**

First, enable JPA auditing by adding the ‘@EnableJpaAuditing’ annotation to your main application class.

package com.example.employeemanagementsystem;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.data.jpa.repository.config.EnableJpaAuditing;

@SpringBootApplication

@EnableJpaAuditing

public class EmployeeManagementSystemApplication {

public static void main(String[] args) {

SpringApplication.run(EmployeeManagementSystemApplication.class, args);

}

}

**Step 2: Configure AuditorAware**

Create a class that implements ‘AuditorAware’ to provide the current user. For demonstration purposes, we'll return a fixed value. In a production environment, you should integrate this with your security context to retrieve the actual user.

package com.example.employeemanagementsystem.config;

import org.springframework.context.annotation.Configuration;

import org.springframework.data.domain.AuditorAware;

import java.util.Optional;

@Configuration

public class AuditorAwareImpl implements AuditorAware<String> {

@Override

public Optional<String> getCurrentAuditor() {

// Return a fixed username for demonstration purposes

return Optional.of("admin");

}

}

**Step 3: Annotate Entities with Auditing Annotations**

Add the relevant auditing annotations to the ‘Employee’ and ‘Department’ entities.

package com.example.employeemanagementsystem.model;

import jakarta.persistence.\*;

import org.springframework.data.annotation.CreatedBy;

import org.springframework.data.annotation.CreatedDate;

import org.springframework.data.annotation.LastModifiedBy;

import org.springframework.data.annotation.LastModifiedDate;

import org.springframework.data.jpa.domain.support.AuditingEntityListener;

import java.time.LocalDateTime;

@Entity

@Table(name = "employees")

@EntityListeners(AuditingEntityListener.class)

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String email;

@ManyToOne

@JoinColumn(name = "department\_id")

private Department department;

@CreatedBy

private String createdBy;

@CreatedDate

private LocalDateTime createdDate;

@LastModifiedBy

private String lastModifiedBy;

@LastModifiedDate

private LocalDateTime lastModifiedDate;

// Getters and setters...

}

@Entity

@Table(name = "departments")

@EntityListeners(AuditingEntityListener.class)

public class Department {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

@CreatedBy

private String createdBy;

@CreatedDate

private LocalDateTime createdDate;

@LastModifiedBy

private String lastModifiedBy;

@LastModifiedDate

private LocalDateTime lastModifiedDate;

// Getters and setters...

}

With these settings, your application will automatically track the ‘createdBy’, ‘createdDate’, ‘lastModifiedBy’, and ‘lastModifiedDate’ fields for each entity.

**Exercise 8: Employee Management System - Creating Projections**

Projections allow you to retrieve specific fields from entities instead of fetching the entire objects. These can be either interface-based or class-based.

**1. Define Projections**

**Interface-Based Projection:**

Create interfaces to define projections for the ‘Employee’ and ‘Department’ entities.

package com.example.employeemanagementsystem.projection;

public interface EmployeeProjection {

Long getId();

String getName();

String getEmail();

String getDepartmentName();

}

public interface DepartmentProjection {

Long getId();

String getName();

}

**Class-Based Projection:**

Define DTO classes for class-based projections.

package com.example.employeemanagementsystem.dto;

public class EmployeeDTO {

private Long id;

private String name;

private String email;

private String departmentName;

public EmployeeDTO(Long id, String name, String email, String departmentName) {

this.id = id;

this.name = name;

this.email = email;

this.departmentName = departmentName;

}

// Getters and setters...

}

public class DepartmentDTO {

private Long id;

private String name;

public DepartmentDTO(Long id, String name) {

this.id = id;

this.name = name;

}

// Getters and setters...

}

**2. Use Projections in Repository Methods**

**Interface-Based Projection:**

Add methods in your repositories to return interface-based projections.

package com.example.employeemanagementsystem.repository;

import com.example.employeemanagementsystem.model.Employee;

import com.example.employeemanagementsystem.projection.EmployeeProjection;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.data.jpa.repository.Query;

import org.springframework.stereotype.Repository;

import java.util.List;

@Repository

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

// Other query methods...

// Interface-based projection

@Query("SELECT e.id as id, e.name as name, e.email as email, e.department.name as departmentName FROM Employee e")

List<EmployeeProjection> findAllEmployeeProjections();

}

**Class-Based Projection:**

Add methods in your repositories to return class-based projections using constructor expressions.

package com.example.employeemanagementsystem.repository;

import com.example.employeemanagementsystem.model.Employee;

import com.example.employeemanagementsystem.dto.EmployeeDTO;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.data.jpa.repository.Query;

import org.springframework.stereotype.Repository;

import java.util.List;

@Repository

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

// Other query methods...

// Class-based projection using constructor expression

@Query("SELECT new com.example.employeemanagementsystem.dto.EmployeeDTO(e.id, e.name, e.email, e.department.name) FROM Employee e")

List<EmployeeDTO> findAllEmployeeDTOs();

}

**3. Fetching Projections in the Service Layer**

Utilize the projection methods within the service layer.

package com.example.employeemanagementsystem.service;

import com.example.employeemanagementsystem.dto.EmployeeDTO;

import com.example.employeemanagementsystem.projection.EmployeeProjection;

import com.example.employeemanagementsystem.repository.EmployeeRepository;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import java.util.List;

@Service

public class EmployeeService {

@Autowired

private EmployeeRepository employeeRepository;

public List<EmployeeProjection> getAllEmployeeProjections() {

return employeeRepository.findAllEmployeeProjections();

}

public List<EmployeeDTO> getAllEmployeeDTOs() {

return employeeRepository.findAllEmployeeDTOs();

}

// Other service methods...

}

**4. Fetching Projections in the Controller Layer**

Define endpoints to return the data using projections.

package com.example.employeemanagementsystem.controller;

import com.example.employeemanagementsystem.dto.EmployeeDTO;

import com.example.employeemanagementsystem.projection.EmployeeProjection;

import com.example.employeemanagementsystem.service.EmployeeService;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.\*;

import java.util.List;

@RestController

@RequestMapping("/employees")

public class EmployeeController {

@Autowired

private EmployeeService employeeService;

@GetMapping("/projections")

public List<EmployeeProjection> getEmployeeProjections() {

return employeeService.getAllEmployeeProjections();

}

@GetMapping("/dto")

public List<EmployeeDTO> getEmployeeDTOs() {

return employeeService.getAllEmployeeDTOs();

}

// Other endpoints...

}

**Testing Entity Auditing and Projections**

1. **Entity Auditing:**
   * Ensure the ‘createdBy’, ‘createdDate’, ‘lastModifiedBy’, and ‘lastModifiedDate’ fields are accurately populated and updated in the database.
2. **Projections:**
   * Test the endpoints ‘GET /employees/projections’ and ‘GET /employees/dto’ to retrieve data using projections.
   * Confirm that the results contain only the fields specified in the projections.

**Exercise 9: Employee Management System - Customizing Data Source Configuration**

In this exercise, we'll explore configuring Spring Boot to handle multiple data sources and externalize configuration properties.

**1. Spring Boot Auto-Configuration**

Spring Boot simplifies data source setup through auto-configuration. It automatically configures a data source when it finds a database driver in the classpath and relevant configuration properties.

**Default Data Source Configuration:**

You can specify the default data source configuration in the ‘application.properties’ file as follows:

# Default H2 Data Source Setup

spring.datasource.url=jdbc:h2:mem:testdb

spring.datasource.username=sa

spring.datasource.password=password

spring.datasource.driver-class-name=org.h2.Driver

spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

spring.jpa.hibernate.ddl-auto=update

**2. Externalizing Configuration**

**Externalize Configuration in ‘application.properties’:**

You can move your data source configurations to the ‘application.properties’ file. Here’s an example with H2 and MySQL data sources:

# Default H2 Data Source Setup

spring.datasource.url=jdbc:h2:mem:testdb

spring.datasource.username=sa

spring.datasource.password=password

spring.datasource.driver-class-name=org.h2.Driver

spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

spring.jpa.hibernate.ddl-auto=update

# MySQL Data Source Setup

app.datasource.mysql.url=jdbc:mysql://localhost:3306/employee\_db

app.datasource.mysql.username=root

app.datasource.mysql.password=yourpassword

app.datasource.mysql.driver-class-name=com.mysql.cj.jdbc.Driver

**Configuring Multiple Data Sources:**

To manage multiple data sources, configure them as follows:

package com.example.employeemanagementsystem.config;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.context.properties.ConfigurationProperties;

import org.springframework.boot.jdbc.DataSourceBuilder;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.Primary;

import org.springframework.core.env.Environment;

import org.springframework.data.jpa.repository.config.EnableJpaRepositories;

import org.springframework.jdbc.datasource.DataSourceTransactionManager;

import org.springframework.orm.jpa.JpaTransactionManager;

import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;

import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;

import javax.sql.DataSource;

import java.util.HashMap;

@Configuration

@EnableJpaRepositories(

basePackages = "com.example.employeemanagementsystem.repository",

entityManagerFactoryRef = "entityManagerFactory",

transactionManagerRef = "transactionManager"

)

public class DataSourceConfig {

@Autowired

private Environment env;

@Primary

@Bean(name = "dataSource")

@ConfigurationProperties(prefix = "spring.datasource")

public DataSource dataSource() {

return DataSourceBuilder.create().build();

}

@Bean(name = "mysqlDataSource")

@ConfigurationProperties(prefix = "app.datasource.mysql")

public DataSource mysqlDataSource() {

return DataSourceBuilder.create().build();

}

@Primary

@Bean(name = "entityManagerFactory")

public LocalContainerEntityManagerFactoryBean entityManagerFactory() {

LocalContainerEntityManagerFactoryBean em = new LocalContainerEntityManagerFactoryBean();

em.setDataSource(dataSource());

em.setPackagesToScan("com.example.employeemanagementsystem.model");

HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();

em.setJpaVendorAdapter(vendorAdapter);

em.setJpaPropertyMap(hibernateProperties());

return em;

}

@Bean(name = "mysqlEntityManagerFactory")

public LocalContainerEntityManagerFactoryBean mysqlEntityManagerFactory() {

LocalContainerEntityManagerFactoryBean em = new LocalContainerEntityManagerFactoryBean();

em.setDataSource(mysqlDataSource());

em.setPackagesToScan("com.example.employeemanagementsystem.model");

HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();

em.setJpaVendorAdapter(vendorAdapter);

em.setJpaPropertyMap(hibernateProperties());

return em;

}

@Primary

@Bean(name = "transactionManager")

public JpaTransactionManager transactionManager() {

JpaTransactionManager transactionManager = new JpaTransactionManager();

transactionManager.setEntityManagerFactory(entityManagerFactory().getObject());

return transactionManager;

}

@Bean(name = "mysqlTransactionManager")

public DataSourceTransactionManager mysqlTransactionManager() {

DataSourceTransactionManager transactionManager = new DataSourceTransactionManager();

transactionManager.setDataSource(mysqlDataSource());

return transactionManager;

}

private HashMap<String, Object> hibernateProperties() {

HashMap<String, Object> properties = new HashMap<>();

properties.put("hibernate.hbm2ddl.auto", env.getProperty("spring.jpa.hibernate.ddl-auto"));

properties.put("hibernate.dialect", env.getProperty("spring.jpa.database-platform"));

return properties;

}

}

**Switching Between Data Sources:**

Specify which data source bean to use for different repositories or services as needed.

**Exercise 10: Employee Management System - Leveraging Hibernate-Specific Features**

Hibernate offers several advanced features that can enhance the performance and capabilities of your application. Here are some key features to utilize:

**1. Hibernate-Specific Annotations**

Hibernate provides additional annotations for more sophisticated mappings and configurations.

**Example of Using Hibernate-Specific Annotations:**

package com.example.employeemanagementsystem.model;

import jakarta.persistence.\*;

import org.hibernate.annotations.Cache;

import org.hibernate.annotations.CacheConcurrencyStrategy;

import org.hibernate.annotations.CreationTimestamp;

import org.hibernate.annotations.UpdateTimestamp;

import java.time.LocalDateTime;

@Entity

@Table(name = "employees")

@Cache(usage = CacheConcurrencyStrategy.READ\_WRITE)

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String email;

@ManyToOne(fetch = FetchType.LAZY)

@JoinColumn(name = "department\_id")

private Department department;

@CreationTimestamp

private LocalDateTime createdDate;

@UpdateTimestamp

private LocalDateTime lastModifiedDate;

// Getters and setters...

}

* ‘@Cache’: Sets up caching for the entity to improve performance.
* ‘@CreationTimestamp’ and ‘@UpdateTimestamp’: Automatically handle timestamps for entity creation and modification.

**2. Configuring Hibernate Dialect and Properties**

Setting the Hibernate dialect ensures compatibility with your specific database.

**Hibernate Configuration in application.properties:**

# Hibernate Settings

spring.jpa.hibernate.ddl-auto=update

spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect

spring.jpa.properties.hibernate.format\_sql=true

spring.jpa.properties.hibernate.use\_sql\_comments=true

spring.jpa.properties.hibernate.show\_sql=true

These properties control the SQL generation and formatting by Hibernate.

**3. Batch Processing**

Batch processing optimizes performance for bulk operations.

**Enable Batch Processing:**

Set batch processing options in ‘application.properties’:

# Hibernate Batch Processing Configuration

spring.jpa.properties.hibernate.jdbc.batch\_size=20

spring.jpa.properties.hibernate.order\_inserts=true

spring.jpa.properties.hibernate.order\_updates=true

**Implement Batch Processing:**

Use batch operations for handling multiple records efficiently in your service layer.

package com.example.employeemanagementsystem.service;

import com.example.employeemanagementsystem.model.Employee;

import com.example.employeemanagementsystem.repository.EmployeeRepository;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import jakarta.transaction.Transactional;

import java.util.List;

@Service

public class EmployeeService {

@Autowired

private EmployeeRepository employeeRepository;

@Transactional

public void saveAllEmployees(List<Employee> employees) {

employeeRepository.saveAll(employees);

}

// Other service methods...

}

Batch processing reduces the number of database interactions needed for bulk operations, leading to better performance.

**Testing Data Source Configuration and Hibernate Features**

1. **Data Source Configuration:**
   * Ensure the application correctly connects to and utilizes multiple data sources.
   * Test CRUD operations across all configured data sources.
2. **Hibernate Features:**
   * Confirm that entity timestamps (‘createdDate’ and ‘lastModifiedDate’) are updated automatically.
   * Validate caching by observing fewer queries to the database.
   * Test batch processing by saving or updating several records and note any performance gains.